ARL-TR-8248 ● Dec 2017

US Army Research Laboratory

Cloud Migration Experiment Configuration and Results

by Michael De Lucia, Justin Wray, and Steven S Collmann

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

DESTRUCTION NOTICE—For classified documents, follow the procedures in DOD 5220.22-M, National Industrial Security Program Operating Manual, Chapter 5, Section 7, or DOD 5200.1-R, Information Security Program Regulation, C6.7. For unclassified, limited documents, destroy by any method that will prevent disclosure of contents or reconstruction of the document.

ARL-TR-8248 ● DEC 2017

US Army Research Laboratory

Cloud Migration Experiment Configuration and Results

by Michael De Lucia
Computational and Information Sciences Directorate, ARL

Justin Wray and Steven S Collmann
ICF International, Columbia, MD

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
December 2017	Technical Report	1 June 2017–10 Oct 2017

4. TITLE AND SUBTITLE
Cloud Migration Experiment Configuration and Results

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Michael De Lucia, Justin Wray, and Steven S Collmann

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Research Laboratory
Computational and Information Sciences Directorate (ATTN: RDRL-CIN-D)
Aberdeen Proving Ground, MD 21005

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-8248

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
ASD(R&E)
Suite 17C08, 4800 Mark Center Drive, Alexandria, VA 22350
CERDEC
Bldg 6003 Combat Drive, APG, MD 21005

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The cloud environment leverages many fundamental technologies. One such technology is virtualization (hypervisor). At a high level, the hypervisor allows for a number of virtual machines to share the physical resources of a single physical machine. A large number of physical machines with hypervisors (host machines) could be networked together to form what is known as a cloud environment. A virtual machine that is hosted on a hypervisor is often referred to as a guest virtual machine. The increase of a number of organizations leveraging the same physical host hypervisor for guest virtual machines opens up the opportunity for what is often referred to as side channel attacks. A technique that allows a virtual machine to accommodate increased resource needs or possibly defend itself from side channel attacks is migration. Migration is the process of moving a guest virtual machine from one physical host to another. This report discusses the different migration types and the results from experimentation in the US Army Research Laboratory Cloud Testbed (Kraken).

15. SUBJECT TERMS
cloud migration, virtual machine migration, hypervisor, cloud environment, virtual machine

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Michael De Lucia
Unclassified	Unclassified	Unclassified	UU	30	19b. TELEPHONE NUMBER (Include area code) (410) 278-6508

Contents

List of Figures

List of Tables

1. Introduction

The cloud environment leverages many fundamental technologies. One such technology is virtualization (hypervisor). At a high level, the hypervisor allows for a number of virtual machines to share the physical resources of a single physical machine. A large number of physical machines with hypervisors (host machines) could be networked together to form what is known as a cloud environment. Although there are many additional components that make up a cloud environment, these are not relevant in this experiment. A virtual machine that is hosted on a hypervisor is often referred to as a guest virtual machine. The increase of a number of organizations leveraging the same physical host hypervisor for guest virtual machines opens up the opportunity for what is often referred to as side channel attacks. An example of a side channel attack within the cloud environment is the observation of a program's memory access patterns to the shared processor cache. A technique that allows a virtual machine to accommodate increased resource needs or possibly defend itself from side channel attacks is migration. Migration is the process of moving a guest virtual machine from one physical host to another.

There are 3 migration types that will be considered for the purposes of this experiment. The 3 types are offline migration, live migration, and cloning migration. The purpose of the migration experiment was to demonstrate the differences in migration types. In order to experiment with the different migration types, a Denial of Service (DoS) scenario was created to increase the resource needs of a host to cause a migration from one physical host to another. In this experiment, the migrations were automatically initiated with an automation package called Terraform. Although these migrations were automated with Terraform, they are normally performed manually. However, the controller would evacuate the underlying node, initiating a migration, if the performance suffered to an unsustainable level.

2. Experiment Overview

This experiment was conducted in the US Army Research Laboratory (ARL) Cloud Testbed (Kraken), which is composed of an OpenStack instantiation. OpenStack is a prevailing cloud infrastructure management software that leverages Kernel Virtual Machine (KVM) as the virtualization layer. The attacker guest virtual machines were a default installation of Kali Linux, and the DoS target guest virtual machine was a standard installation of Ubuntu 16.04 LTS Linux with Apache, MySQL, and PHP, to emulate a real-world web server. The migration was initiated

after the host performance was degraded as a result of the DoS. The environment was set up and configured as shown in Fig. 1.

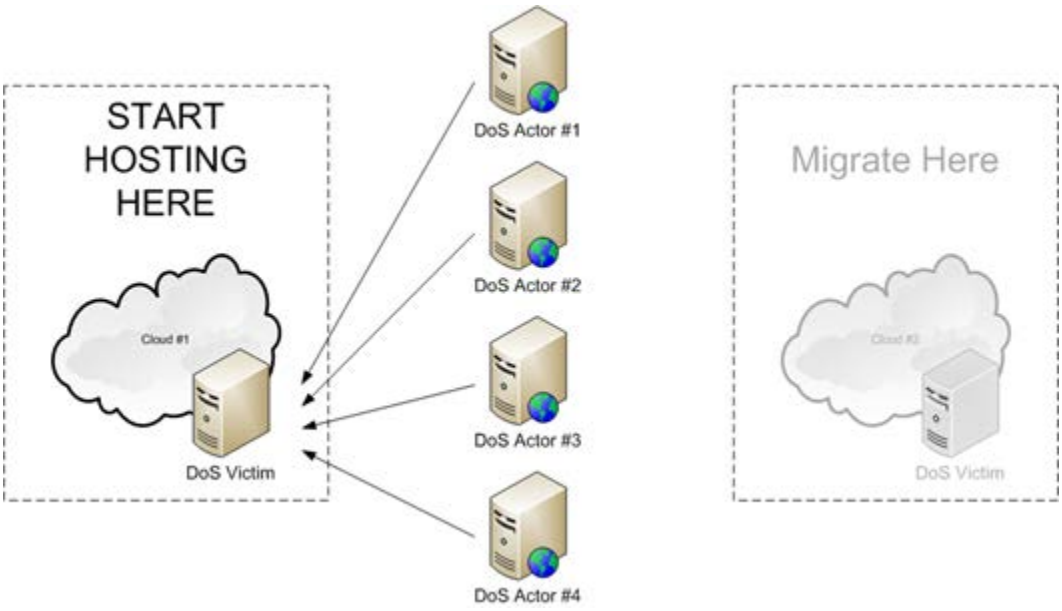

Fig. 1 Experiment setup

The configuration of the environment was built based on Terraform, to provide repeatability of the initial deployment and to ensure an exact replication of the environment for each migration type and test run. Terraform is a cloud management (DevOps) framework that provides "infrastructure as code". Essentially, Terraform allows you to build configuration templates for environments. These templates can easily be shared to allow for identical deployments in other environments such as AWS or VMWare. Terraform is also capable of migration, within a cloud provider or even across providers. All of the migration processes were coded in advance with Terraform to provide repeatability of the process and remove human reaction time from the resulting data collection. The migration process was triggered as the resource consumption rose above a set threshold. As previously stated, the experiment was set up to mimic a migration of a DoS victim virtual machine. The cloud environment was composed of a total of 5 guest virtual machines, consisting of a single DoS victim web server and 4 DoS attackers.

3. Terraform Configuration: General

To facilitate this experiment on the Kraken infrastructure, a base configuration template was created. Terraform uses a separate configuration for each deployment environment. Configurations are textual and can be supplied to Terraform in a number of ways. Primarily, Terraform leverages configuration files to build out an

environment. Given the static nature of most Kraken-related configurations, it is optimal to separate out the configuration into smaller configuration files to be reused across various deployment environments.

When using multiple configuration files, the static files that do not change across experiments can be linked to each environment/experiment configuration directory. This linkage process ensures that if a Kraken-related parameter needs to be modified, the change may be made once it is in the single file, which will impact all of the Terraform configurations.

An important aspect about Terraform configurations is the ordering of the directives (commands). Terraform sequentially processes the configuration files; therefore, dependencies must be configured first. This configuration processing and ordering becomes important when using multiple files. It is essential to name the configuration files in a manner that allows processing in the correct order. Terraform will use an alphanumerical sort to process the files. As such, only a single Terraform configuration should be present in a given directory; therefore, each experiment should have its own terraform configuration directory.

Terraform provides comprehensive documentation on their website, which can be found at https://www.terraform.io/docs/index.html.

Specific Terraform configuration files to enable this experiment can be found in Appendixes A and B.

4. Experiment/Environment Deployment and Execution

Once the environment configuration for the experiment is completed, the process to deploy the experiment with Terraform is straightforward. First, execute Terraform to build a "plan" of deployment. This portion of the process allows Terraform to test the configuration, ensure that it can complete the task, and build the plan of action. This planning process also provides an opportunity for the experiment user to verify the configurations. The *terraform plan* command is used to initiate the planning process. Second, if the configurations are valid and the plan matches the expectations, it can be executed with the *terraform apply* command. The *terraform destroy* command is used to terminate the entire experiment. The following is a summary of the basic Terraform commands.

- *terraform plan* - Initiate the planning process
- *terraform apply* - Execute the experiment plan
- *terraform destroy* - Terminate the entire experiment plan

It is important to note that Terraform maintains a state of the experiment environment, and changes can be made to the environment without the need to first terminate. Simply make the desired changes to the configuration files, re-execute the plan, and apply the respective commands to update the existing deployment.

After the experiment was designed, developed, and configured, the test procedure was automated in the following processes and repeated 10 times for each migration:

- Deploy Experiment Environment
- Initiate Migration Process
- Migration Completes
- Collect and Record Time Duration (using Kraken logs and Unix time tool)
- Destroy Experiment Environment

5. Experiment Results

5.1 Clone Results

The DoS victim guest virtual machine was cloned with the resulting time. For this migration type, the completion time frame and guest virtual machine downtime are the same. A plot of the migration time for 10 tests is shown in Fig. 2. Table 1 calculates a statistical analysis over all 10 test cases, while Table 2 displays the actual migration times.

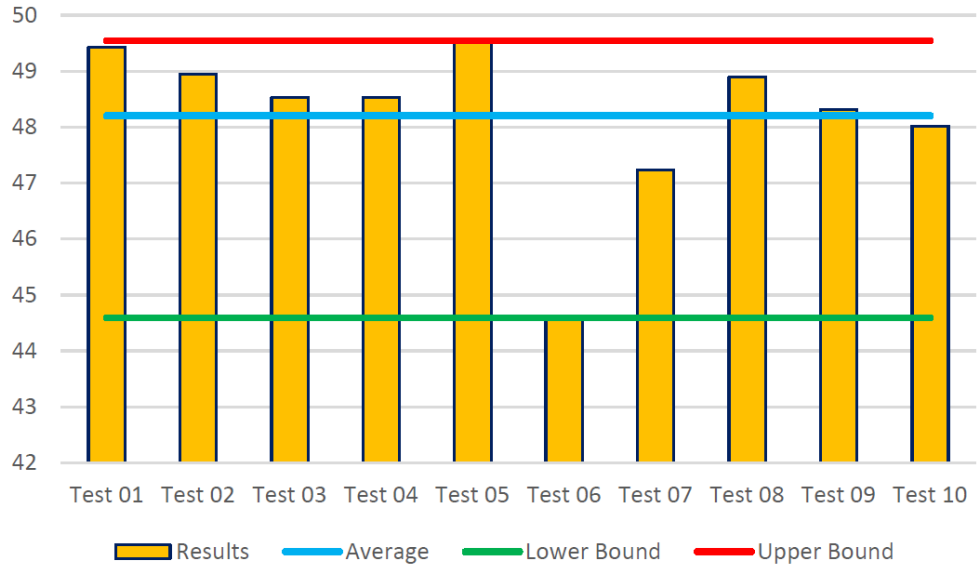

Fig. 2 Clone migration time (s)

Table 1 Statistical analysis

	Time (s)
Average Deviation	0.95
Average (Mean)	48.20
Average (Median)	48.54
Average (Mode)	48.54
Range	44.59–49.55
Variance	1.86
Standard Deviation	1.36
Quartile (25%)	47.82
Quartile (50%)	48.54
Quartile (75%)	49.06

Table 2 Raw data

Test no.	Time (s)
01	49.43
02	48.94
03	48.54
04	48.54
05	49.55
06	44.59
07	47.23
08	48.90
09	48.31
10	48.02

5.2 Offline Migration Results

The DoS victim guest virtual machine was migrated by the controller, by first being taken offline (shutdown) and then being moved before being brought back online. For this migration type, the completion time frame and guest system downtime are the same. A plot of the migration time for 10 tests is shown in Fig. 3. Table 3 calculates a statistical analysis over all 10 test cases, while Table 4 displays the actual migration times.

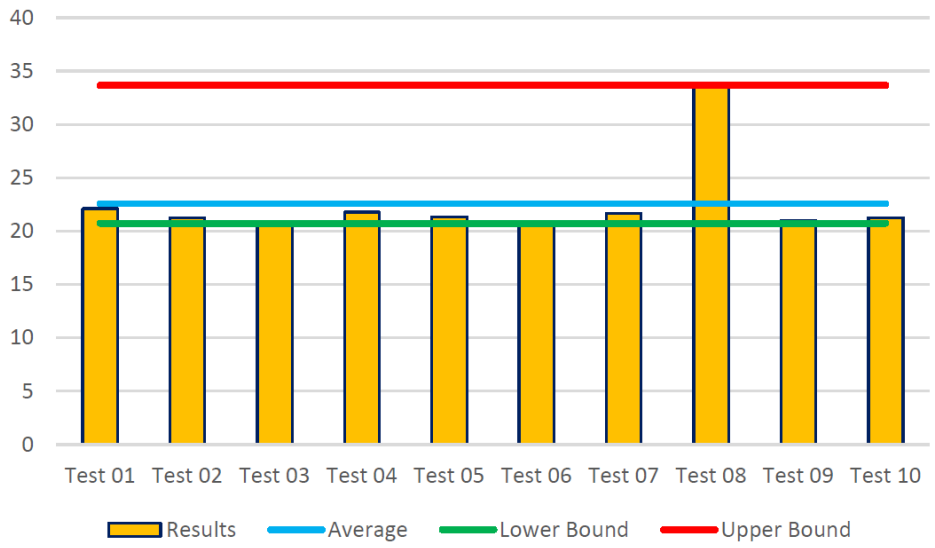

Fig. 3 Offline migration time (s)

Table 3 Statistical Analysis

	Time (s)
Average Deviation	2.22
Average (Mean)	22.57
Average (Median)	21.29
Average (Mode)	N/A
Range	20.73–33.66
Variance	13.82
Standard Deviation	3.72
Quartile (25%)	20.97
Quartile (50%)	21.29
Quartile (75%)	21.86

Table 4 Raw data

Test no.	Time (s)
01	22.11
02	21.27
03	20.89
04	21.78
05	21.31
06	20.73
07	21.68
08	33.66
09	21.00
10	21.26

5.3 Live Migration Results

The DoS victim guest virtual machine was migrated by the controller using block-based live migration. The controller copied the DoS victim guest virtual machine storage disk to the new node before moving the DoS victim guest virtual machine. Once the DoS victim guest virtual machine was running on the new node, the copy on the old node was destroyed. This provides near-zero downtime for the DoS victim guest virtual machine. The resulting time before the guest system was fully migrated was measured. Guest system downtime was also measured and provided separately. A plot of the migration time and the guest down time for 10 tests is shown in Figs. 4 and 5, respectively. Tables 5 and 6 calculate a statistical analysis over all 10 test cases for the migration and guest down times, while Tables 7 and 8 display the actual migration times and guest down times.

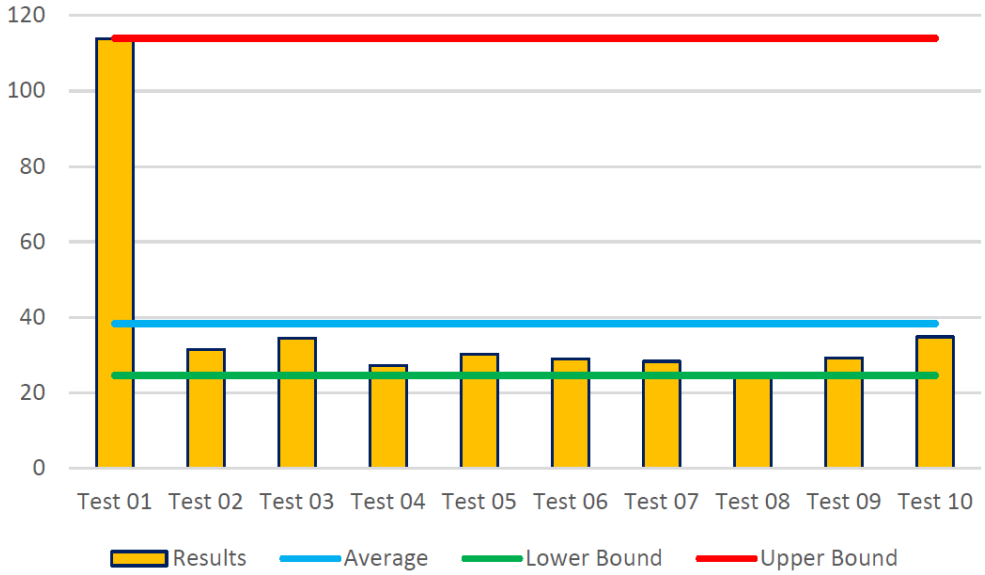

Fig. 4 Live migration completion time (s)

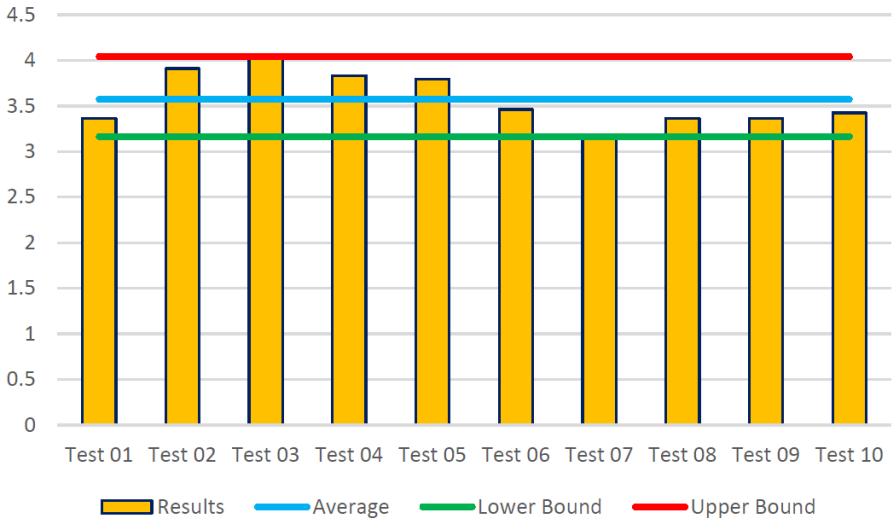

Fig. 5 Live migration guest virtual machine down time (s)

Table 5 Statistical analysis completion time

	Time (s)
Average Deviation	15.10
Average (Mean)	38.36
Average (Median)	29.70
Average (Mode)	N/A
Range	24.61–113.88
Variance	642.44
Standard Deviation	25.35
Quartile (25%)	28.13
Quartile (50%)	29.69
Quartile (75%)	34.63

Table 6 Statistical analysis guest down time

	Time (s)
Average Deviation	0.26
Average (Mean)	3.57
Average (Median)	3.44
Average (Mode)	N/A
Range	3.17–4.04
Variance	0.078
Standard Deviation	0.28
Quartile (25%)	3.36
Quartile (50%)	3.44
Quartile (75%)	3.85

Table 7 Raw data completion time

Test no.	Time (s)
01	113.88
02	31.52
03	34.56
04	27.28
05	30.17
06	29.06
07	28.42
08	24.61
09	29.21
10	34.86

Table 8 Raw data guest down time

Test no.	Time (s)
01	3.37
02	3.91
03	4.04
04	3.83
05	3.79
06	3.46
07	3.17
08	3.36
09	3.37
10	3.42

5.4 Result Summary

5.4.1 Clone Migration

Migration using the cloning methodology had the largest average (mean) time to completion, but it also had the lowest variance. The cloning migration provides more consistency across migration attempts compared to the other migration types. An added benefit to the cloning migration type is the retention of the guest system image at the point in time the migration was started. Although this type has a distinct benefit that is not found in the other migration types, it is the slowest and has the longest impact on the guest system with extended downtime.

5.4.2 Offline Migration

Migration using the offline methodology has the shortest average (mean) time to complete. This offline migration type is also very consistent with a single outlier increasing the variance and upper bound. Although this type is not as consistent as the cloning methodology, it still provides a fairly predictable completion time. Given the short time frame to get the system back online, this migration type offers very little impact to the guest system with a short amount of downtime.

5.4.3 Live Migration

Migration using the live methodology has an average (mean) time to complete in-between the cloning migration and offline migration types; however, live migration provides the lowest amount of guest system downtime. In fact, live migration provides a substantial improvement compared to the downtime experienced across the other migration types. Similar to the offline migration type, live migration also

has a single outlier that increases the upper bound and variance. When removing the outlier, the variance is low and provides a consistent and predictable time to completion; furthermore, the outlier did not impact downtime. As such, this live migration type provides the least variance to overall guest system downtime.

6. Conclusion

Each migration type provides different benefits, guest system downtime, and total time to completion; as such, the specific migration chosen will vary depending on the goals of the migration. However, given the results, we can draw the following conclusions.

If the lowest impact to the guest systems availability is the primary concern during migration, the "Live Migration" type offers the best results. In this case, the guest system remains online for the majority of the migration process, experiencing only a brief lapse in availability.

If retaining a copy of the guest system's state before migration is the primary concern, the "Clone Migration" type offers the only solution. In this case, the guest system experiences the greatest impact to availability, but a copy of the guest system is retained permanently.

Under no circumstances does an "Offline Migration" type provide superior results to either of the other migration types tested. Although the "Offline Migration" provides less impact to the guest system's availability compared with the "Cloning Migration" type, it does so without retaining a copy of the guest system's state and at a substantial increase to availability compared with the "Live Migration"; that is, the "Offline Migration" provides no benefit when compared with "Live Migration". In conclusion, it is likely that the "Live Migration" and "Clone Migration" types are the most beneficial and practical migration types.

INTENTIONALLY LEFT BLANK.

Appendix A. General Terraform Configuration

Authentication Configuration: *C01_auth.tf*

This file should be named *C01_auth.tf*.

The first configuration file provides Terraform with authentication variables that will be used in future configuration parameters.

```
variable "openstack_user_name" {
      description = "OpenStack Username"
      default = "USERNAME"
}

variable "openstack_password" {
      description = "OpenStack Password"
      default = "PASSWORD"
}

variable "openstack_auth_url" {
      description = "OpenStack Authentication URL/API Endpoint"
      default = "http://controller:5000/v2.0/"
}
```

For this authentication configuration file, the only values that must be changed, based on the current Kraken environment, would be the username and password of the user performing the experiment. These authentication values would be based on the OpenStack credentials the experiment user commonly uses, or the global administrator credentials. If the API endpoint or services change in the future, the openstack_auth_url would need to be updated to the new values.

Tenant/Project Configuration: C02_tenant.tf

This file should be named *C02_tenant.tf*.

The second configuration file provides Terraform with project (tenant), network, and availability zone variables that will be used in future configuration parameters.

```
variable "openstack_tenant_name" {
      description = "OpenStack Tenant Name"
      default = "default"
}

variable "tenant_network" {
      description = "OpenStack Neutron Network"
      default = "external"
}

variable "openstack_availability_zone" {
      description = "OpenStack Availability Zone"
      default = "nova"
}
```

Given the current Kraken configuration the only value that must be changed within this configuration file would be the *tenant_network* parameter. However, if in the future multiple projects/tenants or availability zones are created within the Kraken environment the other options may also need to be changed.

It should be noted that the *openstack_tenant_name* variable is synonymous with the project names within OpenStack. In the case that you need to deploy cloud guests to multiple networks or availability zones you can duplicate the above variables and use a different name, such as “tenant_network_02”.

Key Authentication Configuration: C03_keys.tf

This file should be named *C03_keys.tf*.

The third configuration file provides Terraform with the key-pair information to use when deploying cloud guests within the infrastructure. This configuration option sets the keys used for SSH authentication to the guest or the key used to retrieve the Windows password for Windows-based guests. It should be noted that the key-pair must already exist within the underlying hypervisor, in this case OpenStack.

```
variable "openstack_keypair" {
      description = "OpenStack Keypair"
      default = "default"
}
```

If the use of a different key is desired, the key name variable must be updated. Similar to other configuration options, if multiple key usage is desired, the parameter can be duplicated with a different name.

Provider/OpenStack Configuration: P01_openstack.tf

This file should be named *P01_openstack.tf*.

The fourth configuration file provides Terraform with information on which provider (cloud hypervisor) to use for the deployment. In our case, the Kraken infrastructure uses OpenStack.

```
provider "openstack" {
      user_name = "${var.openstack_user_name}"
      password = "${var.openstack_password}"
      auth_url = "${var.openstack_auth_url}"
      tenant_name = "${var.openstack_tenant_name}"
}
```

Given the previous configuration files, the contents within this file should never change when using OpenStack.

If other cloud providers are used in the future, even in combination with OpenStack, this file can be duplicated with the relevant information configured for the other provider. Terraform provides extensive documentation and lists of the available providers within their online documentation, found at https://www.terraform.io/docs/providers/index.html.

Instance/Guest Configuration: Z01_instances.tf

This file should be named *Z01_instances.tf*.

The fifth and final base configuration file provides Terraform with information on what instances to deploy and how to deploy them, within the cloud infrastructure.

```
resource "openstack_compute_instance_v2" "terraform_base" {
      name = "terraform_base"
      image_name = "Base"
      availability_zone  = "${var.openstack_availability_zone}"
      flavor_name = "m1.small"
      key_pair = "${var.openstack_keypair}"
      security_groups = ["default"]
      user_data = "${file("post_deploy.sh")}"
      network {
            name = "${var.tenant_network}"
      }
}
```

This configuration file is the most important to the actual experiment or test being conducted, because this file defines the virtual environment you are deploying. As such, this configuration file is subject to the most change or even duplication, and it should never be linked from a global file.

The first parameter that needs to be changed is the instance name. This must be unique within the configuration and is actually found twice within the configuration: first as the Terraform name on the initial line and second as the name to use within the provider infrastructure (OpenStack, in this case).

The next parameter is extremely important because it defines the image to be deployed. The *image_name* value should be identical to what is found within the provider infrastructure. It should also be noted that the image needs to have already been created for your experiment.

The *availability_zone*, *key_pair*, and network name options should not need to be changed unless duplicates were created in the preceding configuration files.

The *flavor_nam*e option defines the instance resource settings and is based on the naming scheme and configuration of the underlying cloud provider. If your image requires a different resource allocation, this value should be updated accordingly.

The *security_groups* value defines the network access control settings to use for the instance and is again based on the naming scheme and configuration of the underlying cloud provider. These security groups should already be defined within the cloud infrastructure. Should you wish to use the non-default group, which is wide-open/non-filtering, then this value should be updated accordingly.

Finally, the *user_data* parameter allows for a script file to be executed post-deployment. This file should exist in the same directory as the Terraform configuration files and can provide the automation necessary to begin any experimentation processes on the cloud guests.

Global Files/Linking

The following files are likely to be global configuration files and therefore can be configured once and linked (symlink) to the various experimentation directories:

- C01_auth.tf
- C02_tenant.tf
- C03_keys.tf
- P01_openstack.tf

In the case that one, or any, of these files needs to be modified in a specific experiment, that file can be overwritten (non-symlink) with the experiment-specific configuration file. If other providers are configured in the future, those configuration files should also be considered global and linked to the various experiment directories.

Appendix B. Terraform Configuration: Migration Experiment

Although the base configuration was created to rapidly deploy experiments, the vast majority of the Migration Experiment uses the global configurations provided above. The 2 primary configuration files for the migration experiment are the instance configurations: *Z01_instances_victim.tf* and *Z02_instances_attacker.tf*.

Victim Instances: Z01_instances_victim.tf

```
resource "openstack_compute_instance_v2" "experiment_01_victim" {
      name = "experiment_01_victim"
      image_name = "experiment_01_victim"
      availability_zone  = "${var.openstack_availability_zone}"
      flavor_name = "m1.small"
      key_pair = "${var.openstack_keypair}"
      security_groups = ["default"]
      user_data = "${file("post_deploy_victim.sh")}"
      network {
            name = "${var.tenant_network}"
      }
}
```

Attacker Instances: Z02_instances_attacker.tf

```
resource "openstack_compute_instance_v2" "experiment_01_attacker1" {
      name = "experiment_01_attacker1"
      image_name = "experiment_01_attacker"
      availability_zone  = "${var.openstack_availability_zone}"
      flavor_name = "m1.small"
      key_pair = "${var.openstack_keypair}"
      security_groups = ["default"]
      user_data = "${file("post_deploy_attacker.sh")}"
      network {
            name = "${var.tenant_network}"
      }
}

resource "openstack_compute_instance_v2" "experiment_01_attacker2" {
      name = "experiment_01_attacker2"
      image_name = "experiment_01_attacker"
      availability_zone  = "${var.openstack_availability_zone}"
      flavor_name = "m1.small"
      key_pair = "${var.openstack_keypair}"
      security_groups = ["default"]
      user_data = "${file("post_deploy_attacker.sh")}"
      network {
            name = "${var.tenant_network}"
      }
}
```

```
resource "openstack_compute_instance_v2" "experiment_01_attacker3" {
      name = "experiment_01_attacker3"
      image_name = "experiment_01_attacker"
      availability_zone  = "${var.openstack_availability_zone}"
      flavor_name = "m1.small"
      key_pair = "${var.openstack_keypair}"
      security_groups = ["default"]
      user_data = "${file("post_deploy_attacker.sh")}"
      network {
            name = "${var.tenant_network}"
      }
}

resource "openstack_compute_instance_v2" "experiment_01_attacker4" {
      name = "experiment_01_attacker4"
      image_name = "experiment_01_attacker"
      availability_zone  = "${var.openstack_availability_zone}"
      flavor_name = "m1.small"
      key_pair = "${var.openstack_keypair}"
      security_groups = ["default"]
      user_data = "${file("post_deploy_attacker.sh")}"
      network {
            name = "${var.tenant_network}"
       }
}
```

INTENTIONALLY LEFT BLANK.

List of Symbols, Abbreviations, and Acronyms

API	application programming interface
ARL	US Army Research Laboratory
DoS	Denial of Service
KVM	Kernel Virtual Machine

1 (PDF) DEFENSE TECHNICAL INFORMATION CTR DTIC OCA

2 (PDF) DIR ARL IMAL HRA RECORDS MGMT RDRL DCL TECH LIB

1 (PDF) GOVT PRINTG OFC A MALHOTRA

1 (PDF) DIR ARL RDRL CIN D M DE LUCIA